Deep Learning Through the Lens of Classical SQL

Len Du
Australian National University
len.du.public@gmail.com

Abstract

In-database machine learning has been very popular, almost being a cliche. However, can we do it the other way around? In this work, we say "yes" by applying plain old SQL to Deep Learning (DL), in a sense, hypothetically implementing deep learning algorithms with SQL.

Most deep learning frameworks, as well as generic machine learning ones, share a de facto standard of multidimensional array operations, underneath fancier infrastructure such as automatic differentiation. As SQL tables can be regarded as generalizations of (multi-dimensional) arrays, we have found a way to express common deep learning operations in SQL, encouraging a different way of thinking and thus potentially novel models. In particular, one of the latest trend in deep learning was the introduction of sparsity in the name of Graph Convolutional Networks (GCNs), whereas we take sparsity almost for granted in the database world.

As both databases and machine learning involve transformation of datasets, we hope this work can inspire further works utilizing the large body of existing wisdom, algorithms and technologies in the database field to advance the state-of-the-art in machine learning, rather than merely integrating machine learning into databases.

1. Introduction

Both machine learning and databases involve transformation of (or computation over) collections of numbers. Combining the two fields is then an obvious conclusion. But the way of such fusion seems to have been unilateral. Much more effort has been spent towards providing machine learning capabilities in a database context, or so-called "In-Database Machine Learning" [1], compared to integration in the opposite direction, which we call "In-Machine-Learning Database".

We speculate that the connotation of databases has been more towards "systems" than towards algorithms, compared to that of machine learning, making it seemingly more natural to apply the latter to the former. Modern machine learning, in particular deep learning, has been growing into expansive software systems as well, which suggests us to seriously consider the reverse.

In this work we get back at the basic (or not so basic) notion of "transforming collections of numbers" and try substituting the typical operations in machine learning with the most prominent tool in databases, i.e. SQL, to see whatever novel we can find under this different perspective. This paper is mainly to inquire into two research questions: Can we express deep learning in terms of traditional relational database theory, and how to do so if we can? What can we find in traditional relational databases, that could lead to innovations in deep learning? As the information systems community might take a more holistic approach toward the processing of data, including both databases and machine learning, than the more specialized vanilla machine learning community, such inquiry may be of importance to information systems research.

2. Related works

In this section, we review some representative works connecting the two fields of machine learning (in particular, deep learning) and databases, so that we can position this work properly in the whole data science landscape. In particular, reviewing these works helps us with a bird's-eye view of why the relational model, having been ubiquitous in databases since the beginning of the field, should still interest those at the tip of deep learning research.

2.1. Machine learning in databases

MADlib [2] is probably the apex of the classical approach where machine learning subroutines are provided as black-boxes in SQL. MADlib also focuses on conventional machine learning rather than deep

URI: https://hdl.handle.net/10125/71453
978-0-9981331-4-0

learning. Following MADlib, [3] provides a unified architecture for in-database analytics. "Big Data" tools, like Apache Spark and Apache Hadoop, along with their respective machine learning libraries MLlib [4] and Mahout [5], also fall under this category.

SciDB [6, 7] substitutes relational tables with multidimensional arrays. In-database linear algebra and analytics can then be added, resulting in a crossover between a numerical library and a database. Tensor-Relational Model [8] is an elaborated treatise on the role of multidimensional arrays in relational databases. MLog [1] provides a domain-specific language designed for deep learning. The MLog language is integrated into RDBMS by mixing with SQL. It operates on multidimensional arrays (tensors) rather than relational tables. The implementation compiles MLog into TensorFlow[9] programs. In [10], array operations, automatic differentiation and gradient descent are implemented via SQL extensions. [11] envisioned some possible ways to enhance database functionality with deep learning, beyond ease of access of deep learning in databases. SimSQL [12] transpiles SQL instructions to those in Java which then utilize Hadoop to process large amount of data in a distributed fashion.

A strong argument favoring in-database machine learning is that databases are often mature distributed systems, so distributed machine learning would supposedly require little extra effort on the user in a database setting. [13] explores such a setting.

2.2. Database functionality in general machine learning settings

Despite claimed as "an in-database framework", AIDA [14, 15] provides a client interface to a SQL server in the Python language, which is the de facto standard in the machine learning world, AIDA also shifts some of the computation to the server side, or more precisely, a Python interpreter embedded in the database server. So AIDA is best understood as implementing (low-level computation of) machine learning in a database, and then providing the augmented database to machine learning to the user.

ML2SQL [16, 17] compiles a unified declarative domain-specific language to both database operations in SQL and ML-style array operations in python. SystemML [18] and its successor SystemDS [19] also provide a unified language, but they use non-relational databases. TensorLog [20] implements probabilistic logic, essential to probabilistic databases, over typical DL infrastructure.

In addition to machine learning in databases, [11] also envisions providing system-level facilities and distributed computation developed in the database community to deep learning. Finally and perhaps the most well known, the Pandas [21] library familiar to data scientist already provides some essential relational functionalities such as `JOIN` and `SELECT`.

2.3. Neural networks designed for relational models

There are also neural networks specifically designed for learning relations. [22] summarizes very well the effort in this regard before the "deep learning takeover", including Graph Neural Networks (GNN) [23], and Relational Neural Networks [24].

In the more recent surge of deep learning, [25] explores a general deep learning architecture whose outputs are relations, with applications to understanding scenes. [26] combines relational reasoning with recurrent neural networks. [27] further applies the architecture in [25] to complex reinforcement learning tasks. [28] employs a modified logistic regression over hidden layers to learn relations. Lifted Relational Neural Networks [29] combines first-order logic with neural networks to learn relational structures.

Note that all of these models that are *designed to learn relations*, while worth mentioning, overlap little with our claim that relational-model-based SQL can be used as building blocks for *general deep learning*.

2.4. Relational models versus graph convolutional networks

Even being the "fanciest of the fanciest" topic in Machine Learning, Graph convolutional network [30] (GCN) can't escape the link to relational models [31]. [25] advocates relating relational models and graph convolutional networks, as well as deep learning in general, with extensive review.

One interesting fact about GCNs is that GPUs no longer make the usual vast speedups. Even without consideration of relations, GPUs failed to accelerate beyond one order of magnitude [30]. It could be something inherent, given the underlying sparsity, posing the same challenge to deep learning and databases alike.

Apparently, edges of graphs are relations. But relations are not always edges – they could be hyperedges! From the point of view of the "relational" people, it is really a no-brainer that we could have hypergraph variants of GCNs. [32] discusses them without addressing relational models while [33] and [34] address both relational models and hypergraph neural networks.

3. Framework

In this part, we give a big picture of our proposed way of doing deep learning with SQL. While the meaning of deep learning may not be exact enough to prevent intentionally creating a counterexample to our arguments, real instances of deep learning almost universally follow the structures described here, at least in a practical, computational sense.

3.1. The (usual) way of deep learning

A deep learning model can usually be regarded as a scalar function $f(\mathcal{D}, \mathcal{P})$, where $\mathcal{D}$ denotes a set of inputs (data) and $\mathcal{P}$ denotes the model parameters. Our hypothetical goal is to find

$$\operatorname*{argmin}_{\mathcal{P}} f(\mathcal{D}_0, \mathcal{P})$$

where $\mathcal{D}_0$ can be interpreted as either the set of all possible inputs or a test set.

The global minimization is usually intractable. So the learning process involves some iterative optimization involving the gradients $\frac{\partial f(\mathcal{D}, \mathcal{P})}{\partial \mathcal{P}}$. The iterative optimization takes the steps shown in Algorithm 1. This is the most basic pattern. Some deep learning algorithms follow alternative versions. In real world scenarios, it is often only possible to train with stochastic gradient descent which follows the general pattern outlined in Algorithm 2, where only a subset of $\mathcal{D}$ is picked for training each iteration. A recurrent neural network predicting the next symbol given a prefix would be trained with Algorithm 3, in a self-supervised fashion, where the samples are encoded one mega-sequence of vectors $\mathcal{S}$.

Load training set as $\mathcal{D}$;
Randomly initialize $\mathcal{P}$;
while *not meeting stopping criteria* **do**
  (1) Evaluate $f(\mathcal{D}, \mathcal{P})$;
  (2) Evaluate $\frac{\partial f(\mathcal{D}, \mathcal{P})}{\partial \mathcal{P}}$ (taken care of by automatic differentiation; not necessarily mathematically precise) ;
  (3) Update $\mathcal{P}$ with a new value computed with the old $\mathcal{P}$ and $\frac{\partial f(\mathcal{D}, \mathcal{P})}{\partial \mathcal{P}}$ (may carry over state from previous iterations) ;
end

Figure 1. Typical deep learning control flow

Even unconventional uses of deep learning are not so unconventional in terms of control flows. Neural style transfer [35] follows the pattern in Algorithm 4, essentially just switching the argument in Algorithm 1 from $\mathcal{P}$ to $\mathcal{D}$. Note that the content image $\mathcal{I}'$ and style image $\mathcal{I}''$ are never modified once loaded. In practice, it is often possible to leave out $\mathcal{I}'$ in the iterative optimization altogether so long as $\mathcal{I}'$ is used as the initial value of $\mathcal{I}$ [36].

Load $\mathcal{D}$; Initialize $\mathcal{P}$;
while *not meeting stopping criteria* **do**
  (1) Evaluate $f(\mathcal{D}', \mathcal{P})$ where $\mathcal{D}' \in \mathcal{D}$ (chosen per iteration) ;
  (2) Evaluate $\frac{\partial f(\mathcal{D}, \mathcal{P})}{\partial \mathcal{P}}$;
  (3) Update $\mathcal{P}$;
end

Figure 2. Deep learning control flow that stochastic gradient descent uses

Adversarial example generation, like fast gradient sign attack [37], works in a way very similar to neural style transfer by perturbing $\mathcal{D}$ to maximize $f(\mathcal{D}, \mathcal{P})$ in Algorithm 1 rather than perturbing $\mathcal{P}$ to minimize it. The phenomenal generative adversarial network (GAN) pipes two ordinary networks with parameter sets $\mathcal{P}_1$ and $\mathcal{P}_2$ together and run two optimizations in lockstep as shown in Algorithm 5. Function f_1 along with parameters $\mathcal{P}_1$ is the so-called generator network producing "fake" samples given noise as input, while the discriminator network with parameters $\mathcal{P}_2$ trying work out a score for each of both these "fake" samples and the "real" ones given as the training set. Then, one number representing how well the scores separate the two types of samples is summarized from the scores. Finally, the two sets of parameters are optimized with respect to this number, albeit with opposite signs. Surely the order of steps (2) and (3) does not matter.

While there could be other ways to code a deep learning program, the pattern is quite clear. The control flows of deep learning programs are relatively straight-forward, whereas the bulk of the effort are distributed to the design of the models, manifesting primarily in Step (1) of each example, among the 3 major steps conveniently partitioned out of the main loops.

As for Step (2) and (3), there is a separation of concern here. Pragmatically, a major breakthrough that enabled the explosive progress of deep learning is the automatic differentiation. While still an active field of research, development of new deep learning models can be separated from studying automatic differentiation (Step (2)) itself. We can mix and match different flavors

Load $\mathcal{S}$; Initialize $\mathcal{P}$;
while *not meeting stopping criteria* **do**
  (1) Evaluate $f(\mathcal{S}, \mathcal{P})$:
  (1a) Initialize hidden states $\mathcal{H}$ (typically with zeroes);
  (1b) **for** $\mathcal{S}'$ *(embeddings of) sub-sequence of* $\mathcal{S}$ **do**
    (1b1) Evaluate network output (embeddings of predicted sub-sequence) $\mathcal{S}''$ and update hidden states $\mathcal{H}$ with $(\mathcal{S}'', \mathcal{H}) \leftarrow f_1(S', \mathcal{H}, \mathcal{P})$;
    (1b2) Evaluate per-sub-sequence loss $f_2(\mathcal{S}'', \mathcal{S})$ by comparing prediction $\mathcal{S}''$ and the corresponding (embeddings of) sub-sequence of $\mathcal{S}$;
  end
  (1c) compute total loss $f(\mathcal{S}, \mathcal{P})$ by summing or averaging all per-sub-sequence losses ;
  (2) Evaluate $\frac{\partial f(\mathcal{S}, \mathcal{P})}{\partial \mathcal{P}}$;
  (3) Update $\mathcal{P}$;
end

Figure 3. Control flow for training recurrent neural networks

of control flows with different optimization algorithms, or more precisely, different update strategies (Step (3)). While some combinations work better than others, in general inventors of new deep learning models do not concern themselves with which update strategy to pick until tuning the performance of the model.

3.2. Tensor to relations

As we have seen, control flows in deep learning are usually simplistic. While we can never rule something out as non-deep-learning just because it has complex control flows, as a general rule, we could

Load content image $\mathcal{I}'$ and style image $\mathcal{I}''$;
Initialize image $\mathcal{I}$ (maybe randomly but usually $\mathcal{I} \leftarrow \mathcal{I}'$);
Load (pre-trained) $\mathcal{P}$;
while *not meeting stopping criteria* **do**
  (1) Evaluate $f(\mathcal{I}, \mathcal{I}', \mathcal{I}'', \mathcal{P})$;
  (2) Evaluate $\frac{\partial f(\mathcal{I}, \mathcal{P})}{\partial \mathcal{I}}$;
  (3) Update $\mathcal{I}$;
end

Figure 4. Control flow for neural style transfer

Initialize $\mathcal{P}_1$, $\mathcal{P}_2$;
Load true samples $\mathcal{D}$;
while *not meeting stopping criteria* **do**
  (1) Evaluate $f(\mathcal{D}, \mathcal{N}, \mathcal{P}_1, \mathcal{P}_2) = f_2(\mathcal{D}, f_1(\mathcal{N}, \mathcal{P}_1), \mathcal{P}_2)$ where $\mathcal{N}$ is some kind of noise ;
  (2) Evaluate $\frac{\partial f(\mathcal{D}, \mathcal{N}, \mathcal{P}_1, \mathcal{P}_2)}{\partial \mathcal{P}_1}$ and $\frac{\partial f(\mathcal{D}, \mathcal{N}, \mathcal{P}_1, \mathcal{P}_2)}{\partial \mathcal{P}_2}$;
  (3) Update $\mathcal{P}_1$ (maximizing) and $\mathcal{P}_2$ (minimizing) according to respective gradients;
end

Figure 5. Control flow for generative adversarial networks

say that the bulk of deep learning discoveries lie in the transformation of data given the limited variants of control flows. And that transformation might as well be expressed by relational algebra and SQL, with unique benefits.

Modern deep learning infrastructure has been almost universally built upon array-oriented programming paradigms [38, 9, 39]. Looking back, the array-oriented thinking has been in fact deeply rooted in machine learning theories, or more precisely, how those theories have been formulated. Machine learning theories have ubiquitously imposed an implicit order over samples in a dataset by treating a set of vectors as a matrix [40], or a set of n-th order tensors as a $(n+1)$-th order tensor [41], where the tensors serve as no more than the mathematical surrogate of multidimensional arrays for the most of the time [42, 43]. More blatantly, a tensor (in a usual machine learning context) "is a multidimensional array", and "is not to be confused with" those in physics [44]. However, the notation a framework based on (matrix notation or proper tensor calculus) can make a difference in implementation [45, 46].

The ubiquitous use of (ordered) arrays and lack of (unordered) sets mean an implicit constraint on typical machine learning formulations. Namely, the tensors should be arbitrarily permutable along these dimensions. For example, if we are to feed a batch of images to a machine learning algorithm , we should get the same result however we shuffle the order in which the images are lined up (sans floating point errors), but not the order of pixels. Note that this does not apply to online learning where there is indeed a mandated order of samples. There may be situations other than online learning where the order of samples could matter,

especially in some cases where the algorithm is trained "unevenly" favoring early sample to accelerate training, but in general we would expect a machine learning algorithm to output the same result with the order of samples in the same training set (or at least in the same batch) shuffled (with labels shuffled accordingly if applicable, of course).

An obvious justification for the implicit order is the lack of mathematical tools to represent and study sets of objects, compared to well-established theories of matrices and tensors. Imperative languages, which machine learning programs are written in, can represent the same computation in principle. But apparently they are unwieldy if we were to manipulate and deduct theorems about them, nor do they deal with sets natively.

Yet we have had the appropriate tool right here in databases for 50 years [47], where the same frugalness as to avoiding unintentional ordering as we just practiced led to the invention of the very foundation of the field. Both theory[47] and practice [48] of relational databases are built upon the notion of sets.

The array model has been closer to computer hardware and thus led to easier and more performant implementations. But looking forward, we always expect models to catch concepts as precisely as possible, while shifting the work from human to computers as much as possible, especially if we are talking about artificial intelligence! Conceptually, we need to turn from tensors to relations, which we will explore with concrete examples next.

4. DL-in-SQL by Examples

While we are far from a formal proof that SQL can express every possible deep learning model because of the fluidity of the very concept, we nevertheless demonstrate how the bread-and-butter constructs of deep learning can be expressed in SQL.

Here, we use an example deep learning task in stark contrast to typical database-related ones, to demonstrate that our architecture is really geared towards deep learning in general. Meanwhile, we demonstrate how layers frequently employed can be beautifully cast into standard SQL.

4.1. Image classifier convolutional networks

In this example, we demonstrate how to specify a Convolutional Neural Network (CNN) for computer vision in SQL. The model takes N sample images together as input, where N varies depending on how the model is used. The model is fixed for 10 classes, and 32×32 RGB images. Computationally, the neural network displays the structure illustrated in Figure 6. To make things crystal clear, we have drawn all parameters of the network explicitly, so each "step of computation" in a box does not contain any states. This is quite different from many illustrations found elsewhere. For instance, in deep learning jargon, the first "convolutional layer" would conceptually include both "conv1" in the box and the two parameter arrays (kernel and biases) marked in blue, usually not shown explicitly in diagrams.

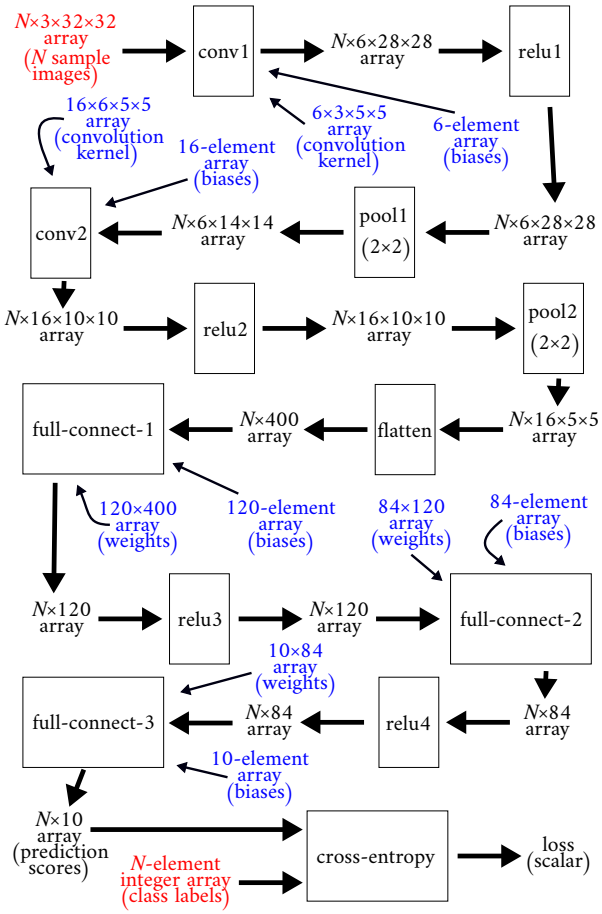

Figure 6. Structure of the CNN for classifying images. Steps of computation are framed, while their inputs and outputs are not. Data (parts of $\mathcal{D}$) are marked in red, while parameters (parts of $\mathcal{P}$) are marked in blue.

Now, we essentially need to express the boxed steps of computation in SQL, with data $\mathcal{D}$ in red and parameters $\mathcal{P}$ in blue given as SQL tables.

First and foremost, let us see what we can do with the convolution step conv1. In the SQL context, we provide the 4-dimensional ($N \times 3 \times 32 \times 32$) array of N sample images as a relation `samples` with the following 5 columns.

```
image, channel, r, c | INTEGER
                 val | REAL
```

The names and types of the columns should be quite self-explanatory. The column `image` refers to indices in the first dimension of the original 4D array, taking the values from 0 to $(N-1)$ (inclusive). Similarly, the column `channel` refers to which one of the three (RGB) channels (2nd dimension), while `r` and `c` refer to which row (3rd dimension) and which column (4th dimension) respectively. The column `val` stores the actual values in the array.

Similarly, the $6 \times 3 \times 5 \times 5$ array of the convolution kernel is presented as a relation `conv1_weight` with 5 columns.

```
out_channel, in_channel, r, c | INTEGER
                       weight | REAL
```

And the biases to the convolutional layer corresponds to a 2-column relation `conv1_bias`.

```
out_channel | INTEGER
       bias | REAL
```

With the input relation ready, we can execute the computation of conv1 with `CREATE TABLE` commands. We do this in two steps. Firstly, we put the results of the *convolution* itself into `conv1_unbiased`.

```
CREATE TABLE conv1_unbiased AS
  SELECT
  image,
  out_channel AS channel,
  samples.r-conv1_weight.r AS r1,
  samples.c-conv1_weight.c AS c1,
  SUM(val * weight) AS val
  FROM
  samples, conv1_weight
  WHERE
  channel = in_channel AND
  r1 BETWEEN 0 AND 32-5 AND
  c1 BETWEEN 0 AND 32-5
  GROUP BY
  image,channel,r1,c1;
```

Then we apply the biases to get `conv1_out`.

```
CREATE TABLE conv1_out AS
  SELECT
  image, channel,
  r1 AS r,
  c1 AS c,
  val + bias AS val
  FROM conv1_unbiased,conv1_bias
  WHERE channel = out_channel;
```

The ReLU layer relu1 is computable by the following.

```
CREATE TABLE relu1_out AS
  SELECT
  image, channel, r, c,
  MAX(0,val) AS val
  FROM conv1_out
```

Executing max-pooling (pool1) is also straight-forward.

```
CREATE TABLE pool1_out AS
  SELECT
  image, channel,
  r/2 AS r,
  c/2 AS c,
  MAX(val) AS val
  FROM relu1_out
  GROUP BY image,channel, r, c;
```

Now the remaining computation steps up till flatten are almost identical to what we have listed except for the table names and array dimensions. We skip these computations, and assume that we have evaluated the output of pool2. flatten is computable in a similar fashion in which we have computed ReLU.

```
CREATE TABLE flatten_out AS
  SELECT
  image,
  (channel*5+r)*5+c AS i,
  val
  FROM pool2_out;
```

A fully-connected layer like fc1 is treated just like a convolution layer. The weights and biases are put in the SQL context as `fc1_weight` with

```
out_dim, in_dim | INTEGER
         weight | REAL
```

and `fc1_bias` with

```
out_dim | INTEGER
   bias | REAL
```

.

Then we can compute `fc1_out` by applying weights and biases in two consecutive steps.

```
CREATE TABLE fc1_unbiased AS
  SELECT
  image,
  out_dim AS i,
  SUM(val*weight) AS val
  FROM flatten_out, fc1_weight
  WHERE i = in_dim
  GROUP BY image, out_dim;
CREATE TABLE fc1_out AS
  SELECT
  image, i,
  val + bias AS val
  FROM fc1_unbiased,fc1_bias
  WHERE i=out_dim;
```

At this stage we could claim that we have specified the *neural network per se*. It is enough for executing inference. However, for training, we still have to show how to compute cross-entropy.

Cross-entropy loss for one sample of computed label

weights $\mathbf{x}$ whose correct label is l is given by

$$\text{loss}(\mathbf{x}, l) = -x_l + \log(\sum_j \exp(x_j)) \ .$$

And we choose to compute the loss over the N samples as the mean over each sample. Assume the output of full-connect-3 to be `fc3_out`. First we compute the right side of "+" with

```
CREATE TABLE x_ent_losses_r AS
  SELECT
  image,
  LOG(SUM(EXP(val))) AS r
  FROM fc3_out
  GROUP BY image;
```

where `LOG()` and `EXP()` are (element-wise) natural logarithm and exponentiation. The left-hand side is just selecting one of 10 elements, listed as follows.

```
CREATE TABLE x_ent_losses_l AS
  SELECT
  fc3_out.image,
  -val AS l
  FROM fc3_out, labels
  WHERE
  fc3_out.image = labels.image
  AND
  i = label;
```

Then we combine both sides to obtain the loss for each image.

```
CREATE TABLE x_ent_losses AS
  SELECT
  image,
  l+r AS val
  FROM
  x_ent_losses_l
  NATURAL JOIN
  x_ent_losses_r;
```

Finally, the average loss is obtained from the following 1-row table.

```
CREATE TABLE x_ent_loss AS
  SELECT
  SUM(val)/COUNT(val)
  FROM x_ent_losses;
```

Now we have finished specifying a deep learning model entirely in SQL.

4.2. Graph convolutional networks

Now, let us see a basic Graph Convolutional Network (GCN) setup in SQL. This GCN is adapted from a simplified version of that in [49], as a very neat tutorial provided by the same author [50]. We further remove the dropout to simplify things, which moderately increases the computation load and over-fitting without changing major results. While dropout in neural networks [51] has been a great

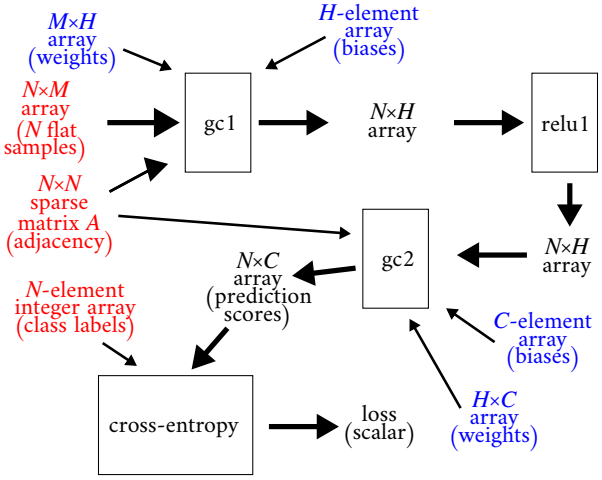

Figure 7. Structure of the exemplar Graph Convolutional Network. Steps of computation are framed, while their inputs and outputs are not. Data (parts of $\mathcal{D}$) are marked in red, while parameters (parts of $\mathcal{P}$) are marked in blue.

invention that certainly affects the actual performance of a neural network, the absence of it rarely renders a network entirely failing to work, especially if the network is not yet so big that it easily over-fits.

The structure of the whole forward computation is as shown in Figure 7. This time we assume N samples of M features to be classified into C classes, with one hidden layer of size H in-between. This structure is actually much simpler than the previous example, the only really new things being the Graph Convolutional layers (gc1 and gc2) and the accompanying $(N \times N)$ adjacency matrix A. So we will focus on them.

From a computational point-of-view, what the Graph Convolutional layer can be considered as two consecutive matrix multiplications plus biasing, despite named "convolutional" layers. That is, the computation before biasing can be simply expressed as AXW , where X denotes the input of the layer and W denotes the weights. Take gc1 for example, X is an $N \times M$ matrix while W is $M \times H$. Then we can add the biases with

$$Y_{i,j} = (AXW)_{i,j} + B_j \ ,$$

for all $i \in \{1..N\}, j \in \{1..H\}$, B being the biases of the layer gc1.

Now continuing with gc1, we assume the following tables in the SQL world.

```
samples(i INTEGER, j INTEGER, val REAL);
gc1_w(i INTEGER, j INTEGER, weight REAL);
gc1_b(i INTEGER, bias REAL);
adj(i INTEGER, j INTEGER, val REAL);
```

Drawing from previous experience of fully-connected layers, we reproduce the above forward computation in SQL with

```
CREATE TABLE gc1_mid AS
  SELECT
  samples.i AS i, gc1_w.j AS j,
  SUM(val * weight) AS val
  FROM samples, gc1_w
  WHERE samples.j == gc1_w.i
  GROUP BY samples.i, gc1_w.j;
```

for the intermediate matrix product XW, and subsequently

```
CREATE TABLE gc1_out AS
  SELECT
  adj.i AS i, gc1_mid.j AS j,
  SUM(adj.val * gc1_mid.val) + bias
    AS val
  FROM adj, gc1_mid, gc1_b
  WHERE
  adj.j == gc1_mid.i AND
  gc1_mid.j == gc1_b.i
  GROUP BY adj.i, gc1_mid.j;
```

for the whole biased output.

The symmetric adjacency matrix A has zeroes for pairs of vertices without an edge in-between and non-zeroes for those connected by an edge. Furthermore, the adjacency matrix is row-normalized from a typical adjacency matrix of 0's and 1's. That is, each row sums to either 1 if there are any edges on the vertex or 0 if the vertex is complete isolated. And so does each column because of symmetry. This adjacency matrix is usually sparse. Or to put it another way, the sparsity is essential to its practical effectiveness, which in turn was probably an important precondition to its current popularity. Yet in the SQL world we do not treat sparsity as something special. Actually we take sparsity for granted. Note that sparsity here refers to the fact that we can easily record, for example, a mapping $\{1 \to 100, 2 \to 500, 65535 \to 100000\}$ in a database without filling the void between 2 and 65535. In some situations, there may be a "sparsity over sparsity" as we may still have missing or null values in a relation, the handling of which is being actively researched. In principle though, such second-order sparsity could always be eliminated by refactoring the relations. For example, if we have a table (a, b, c) where each row may have one or two but not all NULL(s), we can refactor it to six tables (a, b), (b, c), (c, a), (a), (b) and (c) without null values at all. To elaborate, a tuple (A, B, C) where none of A, B and C are NULL, is mapped to one row in each of the six tables, (A, B) in (a, b), (B, C) in (b, c), (C, A) in (c, a), (A) in (a), (B) in (b), and (C) in (c). A tuple $(A, B, \texttt{NULL})$ would be mapped to one row in three tables only, (A, B) in (a, b), (A) in (a), (B) in (b), and absences in the others. A tuple with two NULLs, $(A, \texttt{NULL}, \texttt{NULL})$ would be mapped to (A) in (a), and no records in the remaining five. See [52] for further reference on how to refactoring out NULLs.

While how to create an implementation running as fast as array-basmed deep-learning frameworks is no easy task, we already have battle-hardened semantics and standards in the database world.

5. Discussion

The notion that databases provides the data for dedicated machine learning components to work on has seldom been questioned when unifying the two fields. Now we see we can do it the other way around, interpreting machine learning in terms of database queries.

From a mathematical point-of-view sets are the most basic construct on which vectors, matrices and tensors are constructed. Whereas in terms of programming the situation is reversed, flat dense arrays coming first then sparse structures like sets going after. In this sense, we could consider databases as the deep learning framework in the future which is more conceptually correct, more feature-rich but also more complex and needing more work to optimize. We look at databases, then we know what can be added to deep learning.

And we know we are on the right track looking at some recent progress besides the big trend of graph convolutional networks we have talked about. The Tensor Algebra Compiler [53] stores tensors as trees, arguably reinventing database indexes. Named Tensors [54, 55, 56] where tensor dimensions are assigned alphabetical names, just like good old column headers in SQL, have made their way into PyTorch [38], replaying what happened in the database field after addressing columns with numerical indexes in the very beginning[47]. Alas, proponents of both directions are yet to acknowledge (or discover) the similarity to the existing practices in databases.

The biggest challenge for implementation would be to port automatic differentiation to relational algebra. However, it could also be implemented over yet another layer of flat array (tensor) framework with automatic differentiation, treated as some kind of linear memory to sidestep automatic differentiation from the ground up. Indexes can be built over tensors as well.

Databases have been dealing with sparse data to begin with. While directly run deep learning in database engine may not be competitive as random access too much to be fast at batch processing, we can certainly

continue to bring experience in database to machine learning for a very long time to come.

References

[1] X. Li, B. Cui, Y. Chen, W. Wu, and C. Zhang, "Mlog: Towards declarative in-database machine learning," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1933–1936, 2017.

[2] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, *et al.*, "The madlib analytics library," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, 2012.

[3] X. Feng, A. Kumar, B. Recht, and C. Ré, "Towards a unified architecture for in-rdbms analytics," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, eds.), pp. 325–336, ACM, 2012.

[4] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, pp. 34:1–34:7, 2016.

[5] A. Musselman, "Apache mahout," in *Encyclopedia of Big Data Technologies* (S. Sakr and A. Y. Zomaya, eds.), Springer, 2019.

[6] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, (Berlin, Heidelberg), p. 1–16, Springer-Verlag, 2011.

[7] M. Stonebraker, P. Brown, J. Becla, and D. Zhang, "Scidb: A database management system for applications with complex analytics," *Computing in Science and Engg.*, vol. 15, p. 54–62, May 2013.

[8] M. Kim, *TensorDB and tensor-relational model (TRM) for efficient tensor-relational operations*. Arizona State University, 2014.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[10] M. Schüle, F. Simonis, T. Heyenbrock, A. Kemper, S. Günnemann, and T. Neumann, "In-database machine learning: Gradient descent and tensor algebra for main memory database systems," *BTW 2019*, 2019.

[11] W. Wang, M. Zhang, G. Chen, H. Jagadish, B. C. Ooi, and K.-L. Tan, "Database meets deep learning: Challenges and opportunities," *ACM SIGMOD Record*, vol. 45, no. 2, pp. 17–22, 2016.

[12] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine, "Simulation of database-valued markov chains using simsql," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013* (K. A. Ross, D. Srivastava, and D. Papadias, eds.), pp. 637–648, ACM, 2013.

[13] S. S. Sandha, W. Cabrera, M. Al-Kateb, S. Nair, and M. Srivastava, "In-database distributed machine learning: Demonstration using teradata sql engine," *Proc. VLDB Endow.*, vol. 12, p. 1854–1857, Aug. 2019.

[14] J. V. D'silva, F. De Moor, and B. Kemme, "Making an RDBMS data scientist friendly: Advanced in-database interactive analytics with visualization support," *PVLDB*, vol. 12, no. 12, pp. 1930–1933, 2019.

[15] J. V. D'silva, F. De Moor, and B. Kemme, "Aida: abstraction for advanced in-database analytics," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1400–1413, 2018.

[16] M. E. Schüle, M. Bungeroth, D. Vorona, A. Kemper, S. Günnemann, and T. Neumann, "Ml2sql - compiling a declarative machine learning language to sql and python," in *EDBT*, 2019.

[17] M. Schüle, M. Bungeroth, A. Kemper, S. Günnemann, and T. Neumann, "Mlearn: A declarative machine learning language for database systems," pp. 1–4, 06 2019.

[18] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, *et al.*, "Systemml: Declarative machine learning on spark," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1425–1436, 2016.

[19] M. Boehm, I. Antonov, M. Dokter, R. Ginthoer, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, and B. Rath, "Systemds: A declarative machine learning system for the end-to-end data science lifecycle," 09 2019.

[20] W. W. C. F. Y. Kathryn and R. Mazaitis, "Tensorlog: Deep learning meets probabilistic databases," *Journal of Artificial Intelligence Research*, vol. 1, pp. 1–15, 2018.

[21] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference* (S. van der Walt and J. Millman, eds.), pp. 51 – 56, 2010.

[22] W. Uwents, G. Monfardini, H. Blockeel, M. Gori, and F. Scarselli, "Neural networks for relational learning: an experimental comparison," *Machine Learning*, vol. 82, pp. 315–349, 03 2011.

[23] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[24] W. Uwents and H. Blockeel, "Classifying relational data with neural networks," in *Inductive Logic Programming* (S. Kramer and B. Pfahringer, eds.), (Berlin, Heidelberg), pp. 384–396, Springer Berlin Heidelberg, 2005.

[25] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," in *Advances in Neural Information Processing Systems 30* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), pp. 4967–4976, Curran Associates, Inc., 2017.

[26] A. Santoro, R. Faulkner, D. Raposo, J. Rae, M. Chrzanowski, T. Weber, D. Wierstra, O. Vinyals, R. Pascanu, and T. Lillicrap, "Relational recurrent neural networks," in *Advances in neural information processing systems*, pp. 7299–7310, 2018.

[27] V. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. Reichert, T. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. Botvinick, O. Vinyals, and P. Battaglia, "Deep reinforcement learning with relational inductive biases," in *International Conference on Learning Representations*, 2019.

[28] S. M. Kazemi and D. Poole, "Relnn: A deep neural model for relational learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[29] G. Sourek, V. Aschenbrenner, F. Zelezny, S. Schockaert, and O. Kuzelka, "Lifted relational neural networks: Efficient learning of latent relational structures," *Journal of Artificial Intelligence Research*, vol. 62, pp. 69–100, 2018.

[30] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016.

[31] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*, pp. 593–607, Springer, 2018.

[32] Y. Feng, H. You, Z. Zhang, R. Ji, and Y. Gao, "Hypergraph neural networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3558–3565, 2019.

[33] N. Yadati, M. Nimishakavi, P. Yadav, V. Nitin, A. Louis, and P. Talukdar, "Hypergcn: A new method for training graph convolutional networks on hypergraphs," in *Advances in Neural Information Processing Systems*, pp. 1509–1520, 2019.

[34] J. Jiang, Y. Wei, Y. Feng, J. Cao, and Y. Gao, "Dynamic hypergraph neural networks," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 2635–2641, AAAI Press, 2019.

[35] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Jun 2016.

[36] L. Du, "How much deep learning does neural style transfer really need? an ablation study," in *The IEEE Winter Conference on Applications of Computer Vision (WACV)*, March 2020.

[37] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[39] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[40] C. M. Bishop, *Pattern recognition and machine learning*. springer, 2006.

[41] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[42] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. Phan, "Tensor decompositions for signal processing applications: From two-way to multiway component analysis," *IEEE signal processing magazine*, vol. 32, no. 2, pp. 145–163, 2015.

[43] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *arXiv preprint arXiv:1403.2048*, 2014.

[44] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[45] S. Laue, M. Mitterreiter, and J. Giesen, "Computing higher order derivatives of matrix and tensor expressions," in *Advances in Neural Information Processing Systems*, pp. 2750–2759, 2018.

[46] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "Tensorly: Tensor learning in python," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 925–930, 2019.

[47] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[48] M. Stonebraker and L. A. Rowe, "The design of postgres," *ACM Sigmod Record*, vol. 15, no. 2, pp. 340–355, 1986.

[49] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[50] T. N. Kipf. https://github.com/tkipf/pygcn.

[51] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

[52] C. J. Date, *SQL and Relational Theory - How to Write Accurate SQL Code, Second Edition*. Theory in practice, O'Reilly, 2012.

[53] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[54] A. M. Rush, "Tensor considered harmful," tech. rep., Technical report, Harvard NLP Blog, 2019. Available at http://nlp. seas . . . , 2019.

[55] P. Barceló, N. Higuera, J. Pérez, and B. Subercaseaux, "Expressiveness of matrix and tensor query languages in terms of ml operators," in *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, pp. 1–5, 2019.

[56] P. Barham and M. Isard, "Machine learning systems are stuck in a rut," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 177–183, 2019.